Eliminating Implicit Dependencies in Component Models

Wouter Horré[1], Danny Hughes[1], Ka Lok Man[2], Steven Guan[2], Binbin Qian[2], Tianlin Yu[2], Haofan Zhang[2], Zhun Shen[2], Michel Schellekens[3] and Steve Hollands[4]

[1.] IBBT-Distrinet, Dept. of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium.
[2.] Dept. of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, Suzhou, China.
[3.] Department Computer Science, National University of Ireland, Cork, Ireland
[4.] Synopsys International Limited, Dublin, Ireland

danny.hughes@cs.kuleuven.be

***Abstract*— A software component is defined as a unit of composition with contractually specified interfaces and explicit dependencies that may be independently deployed. Components form generic, re-usable software building blocks, which can be composed into applications and deployed by third parties. A good component model therefore must seek to minimize implicit dependencies in order to maximize re-use and composability. The benefits of component models have led to their widespread application in the area of networked embedded systems and particularly Wireless Sensor Networks. This paper first classifies and analyses the types of dependency that a component may be subject to. Next, we assess the success of contemporary component models in eliminating implicit dependencies and promoting re-usability. We then describe our efforts to reduce implicit distributed dependencies in the design of LooCI: the Loosely-coupled Component Infrastructure. We conclude with a call-to-arms for the component-based software engineering community that suggests avenues for future work.**

Keywords: networked embedded systems; component based software engineering; wireless sensor networks

I. INTRODUCTION

Networked embedded systems such as Wireless Sensor Networks (WSNs) demonstrate a high degree of heterogeneity, complexity and dynamism. Component Based Software Engineering (CBSE) provides mechanisms to manage this heterogeneity and complexity by providing generic and re-usable software building blocks that can be 'composed' together by third parties to form coherent distributed applications. Dynamism may be managed by using runtime reconfiguration to adapt the application to meet changing environmental conditions or user requirements. Szyperski [1] neatly captures the essential features of software components in the following definition: *"a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition"*.

Component interfaces and dependencies are termed 'explicit' because they are contractually specified by the component developer and may be discovered by 3rd parties. Implicit interfaces and dependencies on the other hand are not explicitly specified by the developer and therefore may not be discovered by 3rd parties who thus find it difficult to safely discover and re-use component functionality. Szyperski [1] also highlights that a component should be independently deployable, allowing individual components to be injected into distributed systems as required.

The domain of networked embedded systems has produced a range of component models including: NesC [2], RUNES [3], LooCI [4] and REMORA [5]. In addition, lightweight generic component models such as OpenCOM [6] and OSGi [7] have been used in WSN scenarios such as flood and pollution monitoring [8,3]. However careful analysis reveals that contemporary component models retain implicit dependencies that complicate component re-use and increase the complexity of application composition configuration. In fact, some popular 'component models' do not fit Szyperski's definition very well at all. This paper discusses approaches to minimizing implicit dependencies and suggests directions for future research.

The remainder of this paper is structured as follows: Section II analyses the various dimensions of component dependencies. Section III reviews related work in the area of component models for networked embedded systems. Section IV provides an overview of the Loosely-coupled Component Infrastructure (LooCI), a component model that eliminates distribution dependencies. Section V discusses directions for future work in the area. Finally, Section VI concludes.

II. DIMENSIONS OF COMPONENT DEPENDENCIES

We have identified three types of implicit dependency, which reduce potential for the discovery, and reuse of components by third parties; implicit composition dependencies, implicit distribution dependencies and implicit platform dependencies. The characteristics of each class of dependency are reviewed in sections *II.A* to *II.C* respectively.

A. Composition Dependencies

Eliminating implicit composition dependencies is the raison d'être of component models, so it is to be expected that contemporary component models do well at eliminating this class of dependencies. Contemporary component-based

978-1-4673-0498-6/11/$26.00 ©2011 IEEE

systems [1] ensure composability by eliminating implicit dependencies on other software components, classes and libraries. This allows third parties to safely compose these components into their own application. All software services (*provided interfaces*) and dependencies (*required interfaces or 'receptacles'*) should be specified explicitly in contractual fashion by the developer, and may be inspected by third parties, allowing the application developer to discover, compose and use component functionality without the need to understand the component implementation, which is treated as a black box. Interface definitions may be specified using a specialized language such as IDL [9] as in the CORBA reference model [10] and OpenCOM [6], or language-specific approaches as in NesC [2], RUNES [3], LooCI [4] and REMORA [5]. While all contemporary component models make software dependencies explicit at development-time and thus allow for the static composition of application functionality, only a subset allow this functionality to be inspected and reconfigured at run-time. Introspection (i.e. the ability to inspect components at run-time) and run-time reconfiguration allow third parties and autonomic system elements to dynamically discover and re-use deployed functionality. We argue that run-time discovery and control of component functionality or *dynamic reconfiguration* is an essential requirement for component models that will operate in networked embedded systems as it (a) provides the developer with a mechanism to manage changing environmental conditions and (b) provides a method to efficiently evolve system functionality without resorting to monolithic re-deployment of application functionality, which increases application down-time and power consumption due to unnecessary use of the radio.

B. Distribution Dependencies

For component models that are expected to operate in networked embedded systems such as WSNs, distributed interactions are a critical part of application design and therefore component models should strive to provide a distributed interaction model with no implicit dependencies that allows for safe and reliable third-party re-use of components. *Strictly local* component models [2,3,5] offer no support for distributed interactions. While the developer remains free to use external distribution mechanisms, these are outside of the component model and thus form an implicit dependency, reducing the re-usability of components that include distribution functionality.

The second class of distribution support in contemporary component models is those models that offer transparent distribution using *platform-specific distribution mechanisms* [6,7] such as Java Remote Method Invocation (RMI) [19] or the OSGI service registry [7]. These models are an improvement over strictly local models in the sense that developers may use the constructs of the component model to realize distributed interactions, however they obfuscate the characteristics of distributed interactions. For example; Java RMI depends upon a centralized registry for service discovery and initialization, which introduces an implicit distributed dependency on the node that hosts the registry. As this dependency is not specified in the application composition, the developer cannot discover or reason about it.

The final class of distribution support in contemporary component models are those models that provide *inherently distributed bindings* [4] that do not depend upon external distribution services. In such a model, the application developer explicitly creates all distributed relationships using the same semantics as local component bindings. These models offer the highest level of re-usability for components that embed distributed functionality.

C. Platform Dependencies

Component models should be platform-independent in the sense that components may be deployed on heterogeneous platform configurations, which may evolve over time. In many cases, contemporary component models do not prevent the developer from calling low-level system functionality. This introduces implicit dependencies between components and the expected deployment platform, preventing the safe re-use of components on heterogeneous platforms.

In order to enable safe re-use across heterogeneous platforms, it is critical that component models provide a mechanism to explicitly specify platform dependencies. A variety of techniques have been applied to describe the characteristics of components at development time [22,23], however, current component description schemes have two key shortcomings. Firstly development time platform dependencies are commonly specified in human-readable text, which cannot be parsed in an automated fashion, in turn preventing this information from being discovered at run-time. This shortcoming effectively precludes the safe runtime migration of components across heterogeneous platforms due to *implicit platform dependencies*.

D. Summary of Dependency Types

Table 1: Dependencies in Contemporary Component Models (E = explicit, I = implicit, 'N/P' = not possible)

COMPOSITION DEPENDENCIES	NesC [2]	OpenCOM [6]	OSGI [7]	RUNES [3]	REMORA [5]	LooCI [4]
COMPILE-TIME	**E**	**E**	**E**	**E**	**E**	**E**
RUNTIME	N/P	**E**	**E**	**E**	**E**	**E**
DISTRIBUTION DEPENDENCIES	N/P	I	I	I	I	**E**
PLATFORM DEPENDENCIES						
COMPILE-TIME	**E**	I	I	I	**E**	I
RUN-TIME	I	I	I	I	**E**	I

Table 1 summarizes the explicit and implicit dependencies demonstrated by contemporary component models for networked embedded systems. While this list is not exhaustive we believe that it accurately represents the breadth of work in this area. Section III describes each of the component models listed in Table 1 with particular emphasis on the extent to which they eliminate implicit dependencies.

978-1-4673-0498-6/11/$26.00 ©2011 IEEE

III. Contemporary Component Models

This section reviews related work in the area of component models for networked embedded systems. Section A discusses static component models, which do not provide support for introspection or reconfiguration at runtime. Section B discusses dynamic component models, which allow for introspection and reconfiguration of application compositions at run-time.

A. Static Component Models: NesC

NesC [2] is perhaps the best known and most widely deployed component model for networked embedded systems and is used to implement the TinyOS Operating System [10]. NesC extends the C programming language with an event-driven programming model and mechanisms for explicitly specifying component functionality and dependencies. The NesC extensions eliminate implicit software composition dependencies and thus allow the application developer to compose applications from generic, re-usable building blocks. At compile-time, a NesC application composition is statically optimized and compiled to a monolithic block of executable code, which can neither be inspected nor modified at runtime. Thus, NesC is a *static* component model and provides poor support for scenarios with high levels of dynamism.

In terms of *distribution dependencies*, NesC is a strictly local component model that provides no support for creating distributed relationships. TinyOS [10] provides low-level communication services through the Active Messages paradigm [11], which connects communication and computation by incorporating a reference to a handler component in each message. However, it is not possible to inspect or reconfigure the network configurations at runtime. Therefore Active Messaging results in an implicit distributed dependency. Extensions for TinyOS have been proposed to support Remote Procedure Call (RPC), however, as the extensions do not form part of the NesC model, they result in implicit dependencies [12].

NesC provides good compile-time mechanisms for specifying platform dependencies, however, as NesC system images cannot be inspected or modified after deployment, they become tightly coupled to their host platform and may not be migrated.

B. Runtime Reconfigurable Component Models

This section discusses four key runtime reconfigurable component models: OpenCOM [6], RUNES [3], OSGi [7] and REMORA [5] and critically assesses the degree to which these models have successfully eliminated implicit dependencies.

1) OpenCOM

OpenCOM [6] is a lightweight run-time reconfigurable component model that has been applied to build distributed applications that operate in both networked embedded systems [13] as well as more traditional distributed systems [14]. In terms of *composition dependencies*, OpenCOM allows developers to specify software interfaces and receptacles using a variety of methods including specialized languages such as CORBA IDL [15], the Lorien [16] component definition language and language–specific constructs [13]. These explicit interface and receptacle definitions allow developers to safely discover and re-use third party functionality in both static and dynamic application compositions. The OpenCOM runtime allows the application composition to be inspected and modified after deployment. This effectively eliminates run-time composition dependencies, allowing the developer to safely discover and re-use deployed functionality.

In terms of *distribution dependencies*, OpenCOM is a strictly local component model that provides no support for distributed bindings, however a number of extensions to the OpenCOM core have been proposed to address this. Parlavantis et al. [17] propose a component-based scheme for realizing distributed bindings in which the binding itself is represented as a component. This approach allows for 'pluggable' interaction models, which can be replaced at run-time. However, while the application developer may manually inspect component definitions to learn how a binding component operates, there is no support for discovering the mechanics of how binding components operate at run-time, forming an implicit dependency that limits run-time reconfiguration. More recent OpenCOM-based middleware such as GridKit [13] and MANetKit [18] use the Open Overlays [8] pattern, which provides a standard framework for implementing networking functionality from the transport layer to the application layer. As with the work from Parlavantis et al. [17], Open Overlays allows for the creation of flexible distributed interaction paradigms, while providing enhanced re-usability across different overlay network implementations. However, Open Overlays suffers from the same key problem; components do not make the mechanics of their distribution approach explicit. For example, the version of Open Overlays used in the GridKit WSN middleware [13] uses Java Remote Method Invocation (RMI) [19] to provide transparent distribution, however, this introduces an implicit distributed dependency on the central RMI registry, which forms a single point of failure that is invisible to the application developer.

In terms of *platform dependencies*, OpenCOM does not provide support for the explicit specification of component platform requirements. While some OpenCOM variants such as Lorien [16] allow developers to annotate component descriptions with human readable meta-data, this data is imprecise and optional, which prevents the safe re-use of generic components across heterogeneous platforms.

2) RUNES

The RUNES component model is a branch of OpenCOM [6] that provides specific support for networked embedded systems such as WSN, including additional introspection support in the runtime kernel. As with OpenCOM, RUNES supports the creation of dynamic application compositions, which are free from *composition dependencies* and may be inspected and modified after deployment time. In terms of *distribution dependencies*, RUNES is a strictly local component model, which provides no support for the creation of distributed relationships between components. Instead, developers are expected to implement their own distribution mechanisms as standard RUNES components which exploit lower-level messaging functionality provided by the host OS, or is implemented within the component itself. As this implementation is not explicitly described, an implicit dependency is introduced. In terms of *platform dependencies*,

978-1-4673-0498-6/11/$26.00 ©2011 IEEE

RUNES provides no support for explicitly specifying the platform requirements of components, which prevents components being safely re-used across heterogeneous platforms.

3) Open Services Gateway initiative (OSGi)

The OSGi component model [7] is tightly coupled with the Java programming language and targets powerful embedded devices such as smart phones, web pads and network gateways along with desktop and enterprise computers. OSGi provides a secure execution environment and various supporting services. In terms of *composition dependencies*, OSGi requires that developers explicitly specify component interfaces and receptacles, eliminating implicit composition dependencies at development time. OSGi provides support for modeling components and application compositions using the Service Component Architecture (SCA) [20]. OSGi also supports the run-time inspection and modification of application compositions, eliminating implicit runtime dependencies. In terms of *distribution dependencies*, OSGi supports the transparent creation of remote bindings using distributed OSGi registries [7]. However, the distribution and maintenance of the service registries is managed by the network infrastructure owner and is not explicitly visible to the application developer. This results in implicit distribution dependencies upon the registries, which is a serious problem in unreliable network environments such as WSNs and Mobile Ad-hoc Networks (MANETs). In terms of *platform dependencies*, OSGi allows developers to specify lower-level dependencies in the component meta-data. However, there are two key problems with this approach: firstly OSGi does not provide a machine-readable mechanism for developers to specify platform dependencies and secondly, platform dependency specifications are optional, preventing safe re-use of components.

4) REMORA

REMORA [5] is a component model for networked embedded systems, that provides a single C-like programming language for implementing component functionality and uses the SCA language [20] for specifying component interfaces, receptacles and application compositions. In terms of *composition dependencies*, explicit component specifications, runtime reconfiguration and introspection eliminate composition dependencies at compile time and runtime. In terms of *distribution dependencies*, REMORA is a strictly local component model, which provides no specific support for the creation of distributed relationships, resulting in implicit distribution dependencies. In terms of *platform dependencies*, the remora component language is compiled to work with a standardized Operating System (OS) abstraction layer, which is available for C-based operating systems such as Embedded Linux and Contiki [21]. While the presence of an abstraction layer provides some degree of interoperability, this layer is tightly coupled with C-based Operating Systems, precluding the use of Java-based Operating Systems such as the Sun SPOT [22]. Perhaps more critically, in comparison to platform agnostic component models [3,4,5,6], the OS abstraction layer prevents the developer from using useful platform-specific features. Rather than abstracting over these useful features, we advocate that component models should allow the developer full access to the underlying system, while explicitly exposing these platform dependencies to the application developer.

C. Summary of Related Work

The review of related work performed in this section has revealed that all contemporary component models [2,3,4,5,6] are successful in eliminating *implicit composition dependencies* at development time, however NesC [2], which is the most popular component model for networked embedded systems, does not allow for the inspection and modification of application compositions at runtime, preventing runtime reconfiguration of NesC compositions. As NesC components may not be independently deployed we would argue that they do not fulfill Szyperski's definition of a component [1].

In stark contrast to *composition dependencies*, our analysis reveals that contemporary component models do a poor job of eliminating implicit distribution dependencies. In fact, all of the models reviewed [2,3,5,6] retain implicit distribution dependencies. If one considers that (a) the memory constraints of networked embedded systems enforce small local compositions and (b) the central role of distributed interactions in networked embedded systems, this is clearly a critical shortcoming. Section IV discusses our work in eliminating implicit distribution dependencies in the Loosely-coupled Component Infrastructure (LooCI) [4].

As with distribution dependencies, none of the component models reviewed in this section eliminate explicit platform dependencies [2,3,4,5,6]. This is particularly critical in networked embedded systems, wherein resource constraints require lean operating systems, with non-standard functionality as well as radically heterogeneous hardware platforms. Section V discusses potential approaches to addressing this problem in contemporary component models.

IV. The Loosely-coupled Component Infrastructure (LooCI)

LooCI [4] is a platform-independent component model specifically designed for networked embedded systems. LooCI has been proven to operate effectively even on resource constrained devices such as the AVR Raven (16MHz MCU, 16KB RAM and 48KB Flash). Ports of LooCI are available for the C language on Contiki [21], Java-ME on SQUAWK [24] and Java-SE on OSGi [7]. This demonstrates the portability of LooCI across heterogeneous underlying platforms. As with many embedded Operating Systems [10,21] and component models [2,5], LooCI follows an event-based programming paradigm, which reduces resource contention and is better matched with the characteristics of networked embedded applications such as WSN.

As with all contemporary component models, LooCI eliminates *composition dependencies* at compile time with explicit definitions of interfaces and receptacles. The LooCI runtime also provides support for introspection and reconfiguration. Given the importance of distributed relationships in networked embedded systems we view the

978-1-4673-0498-6/11/$26.00 ©2011 IEEE

elimination of implicit distribution dependencies as a key requirement for LooCI [4]. In our view the successful elimination of distribution dependencies is contingent upon three factors:

(i.) Distributed relationships must not implicitly depend upon functionality deployed on third-party nodes.

(ii.) Distributed relationships must be discoverable and modifiable at runtime.

(iii.) To reduce complexity, the semantics of specifying, inspecting and modifying distributed relationships should match those of local relationships.

In order to realize these objectives, LooCI extends the concept of event-based programming from a local context to a fully distributed context. A decentralized publish-subscribe communication model, or 'event bus', supports distribution. As each node acts as its own broker, the only distributed relationships that exist in a LooCI system are those explicitly specified by the application developer, and thus *distribution dependencies* are effectively eliminated. In order to reduce the complexity of specifying, inspecting and modifying distributed functionality, the semantics of distributed LooCI relationships are identical to local relationships. Details of the LooCI API are provided in [25].

Given the criticality of distributed relationships in networked embedded scenarios, we believe that LooCI offers a significant advance over contemporary components models, however, LooCI does not provide support for explicitly specifying platform dependencies, meaning that components cannot be safely re-used across heterogeneous platforms without specific knowledge of component implementations. We discuss how this need should be addressed in Section V.

V. Discussion

Reviewing the field of contemporary component models has revealed a number of interesting lessons on how composition dependencies, distribution dependencies and platform dependencies can be effectively managed.

Eliminating *composition dependencies* is the raison d'être of contemporary component models and modern component models offer good support for managing composition dependencies at build time and runtime. The lone exception is NesC [2], which offers no support for inspection and reconfiguration of application functionality at run-time.

We have highlighted the relative importance of distributed relationships in networked embedded systems, wherein local resource constraints coupled with complex network environments necessitate distribution support. Despite this clear necessity and numerous applications of component models in networked embedded scenarios [1,8,13,16,18], contemporary component models [2,3,5,6,7] have been unsuccessful in eliminating *distribution dependencies*. In this respect, we believe that LooCI [4] provides a model of how to eliminate distribution dependencies, without introducing additional complexity for the developer.

The only component model reviewed in this paper that eliminates implicit *platform dependencies* is REMORA [5], which achieves this by providing a high-level component implementation language that is compiled to a platform independent virtual machine. However, we advocate against this approach because it abstracts over useful platform-specific features and necessitates the creation of a new abstraction layer for each platform, which incurs significant overhead for the middleware developer.

Rather than abstracting over complexity, we suggest that component models should deal with platform dependencies in the same way that they deal with composition dependencies; by embedding explicit, machine readable descriptions of platform requirements into components. This approach allows component developers to exploit low-level system functionality, while allowing components to be safely reused across heterogeneous platforms. While OSGi [7] provides support for specifying platform dependencies, the fact that such descriptions are optional and not machine-readable limits the usefulness of the approach. We believe that it would be preferable to use an efficient semantic description language to describe platform dependencies such as the one that is used to describe LooCI components [26].

VI. Conclusions

This paper has introduced a classification of dependencies in software component models with three dimensions: composition, distribution and platform dependencies. We reviewed contemporary component models for networked embedded systems and found that while most component models [2,3,4,5,6] eliminate implicit composition dependencies, distribution and platform dependencies remain a problem.

We highlight our work on the LooCI component model [4], which is the first component model to eliminate distribution dependencies in networked embedded scenarios. Based upon our experiences with LooCI, we advocate that distribution should be elevated to a first-class concern for component models and suggest that the component model should allow for the explicit specification of distributed dependencies without introducing implicit relationships to remote nodes.

Along with distribution dependencies, component platform dependencies must also be specified if components are to be safely re-used across heterogeneous hardware and software platforms. We suggest lightweight semantic description technologies [26] as a promising approach to addressing this problem, and advocate that the developers of component models for networked embedded systems pursue research in this area. In summation, we believe that contemporary component models hold great promise, but do not yet live up to their sales-pitch in terms of providing truly generic and re-usable application building blocks. To achieve this vision, will require reconsideration of all classes of component dependency throughout the component lifecycle.

Acknowledgments

This research is partially funded by the Inter-University Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund K.U. Leuven and the Xi'an

Jiaotong-Liverpool University Research Development Fund, under grants 2010-26 and 10-02-03.

REFERENCES

[1] Szyperski C., Component Software - Beyond Object-Oriented Programming. Addison-Wesley / ACM Press, second edition, 2002. With Dominik Gruntz and Stephan Murer.

[2] Gay D., Levis P., Von Behren R., Welsh M., Brewer E., Culler D., The NesC Language: A Holistic Approach to Networked Embedded Systems, in Proc. of the conference on Programming Language Design and Implementation, ACM SIGPLAN 2003, San Diego, California, USA, pp. 1 – 11.

[3] Costa P, Coulson G, Gold R, Lad M, Mascolo C, Mottola L, Picco GP, Sivaharan T, Weerasinghe N, Zachariadis S (2007) The RUNES middleware for networked embedded systems and its ap- plication in a disaster management scenario. In: Proc of the 5th annual IEEE international conference on pervasive computing and communications, PerCom'07, White Plains, New York, pp 69–78

[4] Hughes D, Thoelen K, Horré W, Matthys N, Michiels S, Huygens C, Joosen W (2009) LooCI: a loosely-coupled component infrastructure for networked embedded systems. In: Proc of the 7th international conference on advances in mobile computing & multimedia, MoMM'09. ACM, New York.

[5] A Component-based Approach for Service Distribution in Sensor Networks, Amirhosein Taherkordi, Romain Rouvoy and Frank Eliassen, in 5th international workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens), co-located with ACM/IFIP/USENEX 11th Middleware Conference, Bangalore, India, Dec. 2010.

[6] Coulson G, Blair G, Grace P, Taiani F, Joolia A, Lee K, Ueyama J, Sivaharan T (2008) A generic component model for building systems software. ACM Trans Comput Syst 26(1)

[7] Rellermeyer J., Alonso G., Concierge: A Service Platform for Resource-Constrained Devices, in ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, June 2007, pp. 245 – 258

[8] Grace P., Hughes D., Porter B., Blair G., Coulson G., Taiani F., Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity, in Proc. of the European Conference on Computer Systems (EuroSys'08), Glasgow, Scotland, UK, March 2008, pp. 123-136.

[9] OMG Interface Definition Language (IDL), available online at: http://www.omg.org/gettingstarted/omg_idl.htm [accessed 23/09/11].

[10] Hill J, Szewczyk R, Woo A, Hollar S, Culler D, Pister K (2000) System architecture directions for networked sensors. ACM SIG- PLAN 35(11):93–104

[11] Buonadonna P .,Hill J., Culler D., Active Message Communication for Tiny Networked Sensors, in Proc. of the 20th annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom'01), Anchorage, Alaska, USA, April 2001, pp. 1-11.

[12] May T. D., Dunning S. H., Hallstrom J. O., An RPC Design for Wireless Sensor Networks, in Proc. of the IEEE International Mobile Adhoc and Sensor Systems Conference, (MASS'05), Washington, DC, USA, November 2005, pp. 138-146.

[13] Hughes, D., Greenwood, P., Blair, G.S., Coulson, G., Grace, P., Pappenberger, F., Smith, P., Beven, K., An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring, Concurrency and Computation: Practice and Experience, Vol 20 No 11, pp 1303-1316, 2007.

[14] Lee, K., Coulson, G., "Supporting Runtime Reconfiguration on Network Processors", Journal of Interconnection Networks, Vol 7, No 4, Special Issue on Information Networking and P2P Systems, World Scientific Publishing Co., pp 475-492, 2006.

[15] OMG Interface Definition Language (IDL), available online at: http://www.omg.org/gettingstarted/omg_idl.htm (accessed 22/09/11)

[16] Porter, B., Coulson, G., Roedig, U., "Type-Safe Updating for Modular WSN Software", in proc. Of 7th International Distributed Computing in Sensor Systems (DCOSS), Barcelona, Spain, pp. 1-8.

[17] Parlavantzas N., Coulson G., Blair G., An Extensible Binding Framework for Component-Based Middleware, in Proc. of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), Brisbane, Australia, September 2003, pp 252 - 263.

[18] Ramdhany, R., Grace, P., Coulson, G., Hutchison, D., "MANETKit: Supporting the Dynamic Deployment and Reconfiguration of Ad-Hoc Routing Protocols", in proc. of IFIP/ACM/USENIX Middleware'09, 2009, Urbana Champaign, Illinois, USA, December 2009, pp. 261 – 266.

[19] Java Remote Method Invocation (RMI), available online at: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, (accessed 22/09/11).

[20] Service Component Architecture (SCA), available online at: http://osoa.org/pages/viewpage.action?pageId=46 (accessed 22/09/11)

[21] Dunkels A., Grönvall B., Voigt T., Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors, in Proc. of 29th IEEE International Conference on Local Computer Networks (LCN'04), Tampa, FL, USA, November 2004, pp. 455 – 462.

[22] Qiong W., Jichuan C., Hong m., Fuqing Y., JBCDL: an object-oriented component description language, Technology of Object-Oriented Languages, 1997. TOOLS 24. Proceedings , vol., no., pp.198-205, Sep 1997.

[23] Zhang S., Goddard S., "xSADL: an architecture description language to specify component-based systems," Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on , vol.2, no., pp. 443- 448 Vol. 2, 4-6 April 2005

[24] Simon D., Cifuentes C., Cleal D., Daniels J., White D., Java on the Bare Metal of Wireless Sensor Devices: the Squawk Java Virtual Machine, in Proc. of the 2nd International Conference on Virtual Execution Environments, Ottawa, Canada, June 2006, pp 78 – 88.

[25] The LooCI Project on Google Code, available online at: http://code.google.com/p/looci/ (accessed 29/09/11).

[26] Thoelen K., Matthys N., Horre W., Huygens C., Joosen W., Hughes D., Fang L., Guan S., Supporting Reconfiguration and Re-use through Self-Describing Component Interfaces, in proc. of International Workshop on Middleware for Sensor Networks (MidSens'10), Bangalore, India, 2010.